PageRank, ProPPR, and Stochastic Logic Programs

Dries Van Daele, Angelika Kimmig, and Luc De Raedt

Department of Computer Science, KU Leuven
{dries.vandaele,angelika.kimmig,luc.deraedt}@cs.kuleuven.be

Abstract. A key feature of ProPPR, a recent probabilistic logic language inspired by stochastic logic programs (SLPs), is its use of personalized PageRank for efficient inference. We adopt this view of probabilistic inference as a random walk over a graph constructed from a labeled logic program to investigate the relationship between these two languages, showing that the differences in semantics rule out direct, generally applicable translations between them.

1 Introduction

The need to combine reasoning about relational data and reasoning under uncertainty has led to a variety of languages and formalisms for statistical relational learning [5] and probabilistic logic learning and programming [4], but a deep understanding of the relation between different languages is often lacking. In this paper, we provide a detailed account of the relation between two such languages, namely stochastic logic programs (SLPs) [3, 6], one of the oldest languages in the field, and the recently introduced probabilistic Prolog ProPPR [9, 10], whose semantics is inspired by that of SLPs. Both languages use a labeled logic program to define a probability distribution over all answers to a query, but ProPPR biases this distribution towards answers obtained by shorter derivations in the program. This makes it possible to use variations of the PageRank algorithm [7], originally proposed for ranking web pages, for efficient probabilistic inference. We use this view of inference as a random walk over a graph constructed from a labeled logic program as the formal framework for our comparison, which shows that the differences in semantics rule out direct, generally applicable translations between programs in the two languages.

This paper is structured as follows. We review the background in Section 2. Section 3 focuses on the structure of the graphs on which the random walks are performed. Section 4 discusses the transition probabilities, highlighting their influence on approximate inference.

2 Background

We start by reviewing the background and formal notation.

2.1 PageRank

The PageRank algorithm [7] was originally introduced for the purpose of ranking web pages on the World Wide Web. The *PageRank vector* is a probability distribution computed by applying PageRank on a graph. For each node in the graph, the PageRank vector specifies a probability that expresses its relative importance. Intuitively, this distribution can be regarded as the likelihood that a 'random surfer' [2] arrives at the respective web page. By regarding the World Wide Web as a graph where web pages are nodes and hyperlinks are edges, a 'random surfer' can be simulated by executing a random walk on it. In a random walk on an unweighted, directed graph, one moves from one node to the next by choosing uniformly between outgoing edges. More generally, we can consider a weighted, directed graph where a distribution is defined over the outgoing edges of each node. We can formalise this as a Markov chain whose states are the nodes in the graph, and whose transition matrix S is defined by the probabilities of moving from any node to another. The 'random surfer' model provides an extension to ensure that the random walk can reach every part of the graph. For this purpose, a jump to any node in the graph is introduced that occurs with probability $\gamma \in (0, 1)$. This jump can be represented using a distribution s called the *personalization vector*. When s is not uniformly distributed over all nodes, the algorithm is referred to as personalized PageRank. Given the transition matrix A that captures the behaviour of the 'random surfer' model, i.e. $A = ((1-\gamma)S + \gamma \mathbf{1}s)$ with $\mathbf{1}$ a column vector of ones, the personalized PageRank vector π is defined as its stationary distribution:

$$\pi = A^T \pi$$

The PageRank vector can be computed using a simple iterative method called the power iteration method. Given an initial distribution x_0 over the nodes, the power iteration method computes $x_{k+1} = A^T x_k$ for increasing values of k, until the resulting vector has converged towards the PageRank vector. It thus simulates the flow of probability through the graph until a fixpoint is reached.

2.2 Logic Programming

A *definite clause program* $\mathcal{L}$ (or *logic program* for short) is a set of definite clauses. In a definite clause $a \leftarrow b$, the head a consists of a single atom, and the body $b = a_1, \ldots, a_n$ is a conjunction of atoms. A *substitution* θ is an expression of the form $\{V_1/t_1, ..., V_m/t_m\}$ where the V_i are different variables and the t_i are terms. Applying a substitution θ to a formula f results in the expression $f\theta$ where all variables V_i in f have been simultaneously replaced by their corresponding terms t_i in θ. Two formulas f_1 and f_2 can be *unified* if and only if there are substitutions θ_1 and θ_2 such that $f_1\theta_1 = f_2\theta_2$.

The main inference task in logic programming is to determine whether a given atom, also called *query* (or *goal*), is true in the least Herbrand model of a logic program. If the answer is yes (or no), we also say that the query *succeeds*

```
0.3: q(X) ← r(X), s(X).
0.7: q(X) ← u(X).

0.6: r(a).
0.4: r(b).

0.3: s(a).
0.7: s(b).

0.1: u(a).
0.9: u(b).
```

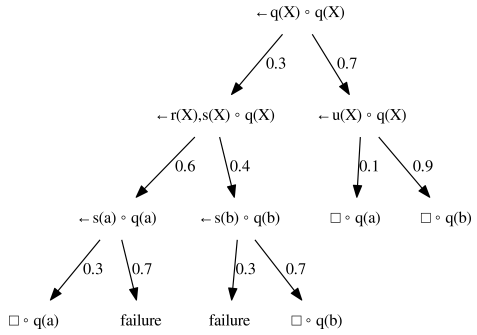


Fig. 1. Example of a pure and complete SLP and its SSLD tree for query $\leftarrow q(X)$

(or *fails*). If such a query is not ground, inference asks for the existence of an *answer substitution*, that is, a substitution that grounds the query into an atom that is part of the least Herbrand model.

Prolog answers queries using *refutation*, that is, the negation of the query is added to the program and resolution is used to derive the empty clause. This process, which continues until the empty goal is reached, can be depicted by means of an *SLD-tree*. The root of such a tree corresponds to the query, each branch to a *derivation*, that is, a sequence of clauses used for resolution steps. To simplify notation when discussing ProPPR later in the paper, the following definition labels each node in the SLD tree with both the subgoal at that node and the query as instantiated by the derivation leading to that node.

Definition 1 (SLD tree). *For a logic program $\mathcal{L}$ and query q, the SLD tree $\mathcal{T}$ is the labeled tree constructed as follows:*

- *The root of $\mathcal{T}$ is labeled $\langle \leftarrow q \circ q \rangle$.*
- *Each node $N = \langle \leftarrow q_1, \ldots, q_n \circ q\theta_N \rangle$ in $\mathcal{T}$ has a child node N_C for every clause $C = h \leftarrow b_1 \ldots, b_m$ in $\mathcal{L}$ whose head h has the same predicate as q_1. If h and q_1 cannot be unified, N_C is labeled with* failure*, and called a* failure node*. Else, let θ be the most general unifier of h and q_1. If the resolvent is the empty clause, N_C is labeled $\langle \square \circ q\theta_N\theta \rangle$, and called a* success node*, else, it is labeled $\langle \leftarrow (b_1, \ldots, b_m, q_2, \ldots, q_n)\theta \circ q\theta_N\theta \rangle$.*

2.3 Stochastic Logic Program

A stochastic logic program (SLP) [3, 6] is a definite clause program whose clauses have a probabilistic interpretation. A *pure* SLP labels each of its clauses with a probability (i.e. $p : a \leftarrow b$ where $p \in [0, 1]$). In a *complete* SLP, the probability labels of the clauses that share the predicate symbol in their head sum up to one. We use the pure and complete SLP in Figure 1 as our running example.

Definition 2 (SSLD tree). *The stochastic SLD tree (SSLD tree) $\mathcal{T}_\mathcal{S}$ for a pure SLP $\mathcal{S}$ and query q is the SLD tree for q and $\mathcal{S}$, where each edge is labeled with*

```
about(X,Z) :- handLabeled(X,Z) # base.
about(X,Z) :- sim(X,Y),about(Y,Z) # prop.
sim(X,Y) :- links(X,Y) # sim,link.
sim(X,Y) :- hasWord(X,W),hasWord(Y,W),linkedBy(X,Y,W) # sim,word.
linkedBy(X,Y,W) :- true # by(W).
```

Fig. 2. Example ProPPR program from [9]; atoms to the right of `#` are rule features.

the probability of the clause used in the corresponding resolution step. $\mathcal{N}_\theta$ is the set of nodes labeled $\langle \square \circ q\theta \rangle$, and $\mathcal{N}$ the union of all such $\mathcal{N}_\theta$. For each node $N \in \mathcal{N}$, we have

$$P_{\mathcal{S}}(N) = \prod_{p_i : C_i \in d(N)} p_i$$

where $d(N) = p_1 : C_1, \ldots, p_n : C_n$ is the derivation ending in N.

In the SSLD tree for our example, cf. Figure 1, the probability of the leftmost derivation $0.3 : q(X) \leftarrow r(X), s(X), 0.6 : r(a), 0.3 : s(a)$ is $0.3 \cdot 0.6 \cdot 0.3 = 0.054$, and similar for all others. Note that several nodes can have the same label (e.g., two nodes are labeled $\langle \square \circ q(a) \rangle$), so we cannot refer to a node by its label only.

Definition 3 (SLP probability $P_{\mathcal{S}}$). *Given a pure SLP $\mathcal{S}$, the probability of a query q with answer substitution θ is the sum of probabilities of all success nodes in $\mathcal{T}_{\mathcal{S}}$ with label $\langle \square \circ q\theta \rangle$, normalized for successful derivations:*

$$P_{\mathcal{S}}(q\theta) = \frac{\sum_{N \in \mathcal{N}_\theta} P_{\mathcal{S}}(N)}{\sum_{M \in \mathcal{N}} P_{\mathcal{S}}(M)}$$

In our example, we get

$$P_{\mathcal{S}}(\mathtt{q(a)}) = (0.054 + 0.07)/(0.054 + 0.07 + 0.084 + 0.63) = 0.148$$
$$P_{\mathcal{S}}(\mathtt{q(b)}) = (0.084 + 0.63)/(0.054 + 0.07 + 0.084 + 0.63) = 0.852.$$

2.4 ProPPR

The probabilistic logic programming language ProPPR [9] is a recent language similar to SLPs, but with a bias towards shorter derivations, which allows for efficient approximate inference. ProPPR does not directly attach probabilities to its clauses. Instead, ProPPR supports a more flexible weighting of clauses through the use of weighted features. Each clause is annotated with one or multiple features, whose weights can be learned from data. An example of some ProPPR rules can be found in Figure 2. Here, we abstract away from the internal structure of the clause labels, and treat them as numerical weights. This abstraction is possible, because in essence, features associate a fixed, numerical weight with each instance of a clause after substitution with the most general unifier during SLD resolution. ProPPR gives special treatment to a set of facts

called the database facts. All database facts have the same default weight w that is not modified during learning. Additionally, the semantics for predicates defined using these facts varies from those defined using normal ProPPR rules. Considering this, we can abstract away from the weighted features, but have to maintain a distinction between predicates defined using standard ProPPR rules, and those defined using database facts. We define a ProPPR program $\mathcal{P}$ as a tuple $(\mathcal{C}_p, \mathcal{C}_{db})$ where $\mathcal{C}_p$ is a set of weighted, definite clauses $w_i : C_i$, and $\mathcal{C}_{db}$ is a set of database facts f_j, all with the same default weight $w_j = w$.

Similar to how an SSLD tree is used to depict the process of deriving answers for a given SLP and query, the process employed by ProPPR can be captured in the PageRank ProPPR graph.

Definition 4 (PageRank ProPPR graph). *For ProPPR program $\mathcal{P}$, query q, restart weight $\alpha \in (0, 1)$, and loop weight $\beta > 0$, the PageRank ProPPR graph $\mathcal{G}_{\mathcal{P},\alpha,\beta,q} = (\mathcal{N}_\mathcal{G}, \mathcal{E}_\mathcal{G})$ is defined as follows. Let $\mathcal{T}$ be the SLD tree for q and $\mathcal{P}$.*

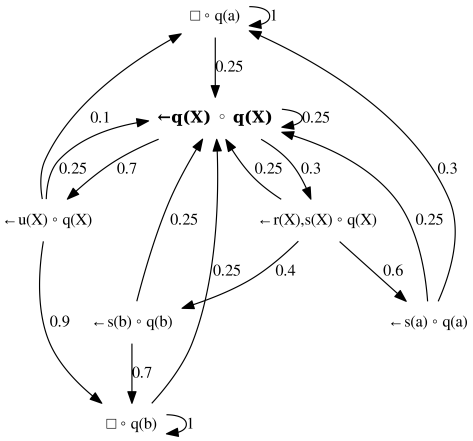
1. *For each label $\langle \leftarrow s \circ q\theta \rangle$ or $\langle \square \circ q\theta \rangle$ appearing in $\mathcal{T}$, $\mathcal{N}_\mathcal{G}$ contains a unique node with that label.*
2. *$\mathcal{E}_\mathcal{G}$ contains an edge (l_1, l_2) if there is an edge between a pair of nodes labeled l_1 and l_2 in $\mathcal{T}$. The edge is labeled with the weight of the clause applied in the corresponding resolution step.*
3. *For every node $N \in \mathcal{N}_\mathcal{G}$, $\mathcal{E}_\mathcal{G}$ contains a restart edge $(N, \langle \leftarrow q \circ q \rangle)$ to the query node with label α.*
4. *For every success node $N = \langle \square \circ q\theta \rangle$, $\mathcal{E}_\mathcal{G}$ contains an edge (N, N) with label β.*

The PageRank ProPPR graph thus does not contain failure nodes. As node labels are unique, we also refer to a node by its label. Figure 3 shows the PageRank ProPPR graph derived from the SSLD tree in Figure 1. In contrast to the labels in an SSLD tree, the labels in a PageRank ProPPR graph do not directly correspond to the transition probabilities of a random walk. ProPPR normalizes these labels to obtain transition probabilities, using a different normalization strategy depending on whether the predicate associated with the node in question is defined in $\mathcal{C}_p$ or $\mathcal{C}_{db}$. If the predicate is defined in $\mathcal{C}_p$, then the distribution over the outgoing edges is determined by normalizing over all outgoing edges. If it is a database fact defined in $\mathcal{C}_{db}$, the weight of the restart edge remains unchanged, and the remaining outgoing edges are normalized to add up to $1 - \alpha$. Figure 4 shows the PageRank ProPPR graph in Figure 3 with edges labeled with the transition probabilities obtained through normalization. Note that all the predicates in the example are defined in $\mathcal{C}_p$.

The loop weight β for the PageRank ProPPR graph in Figure 3 is set to 1. This ensures that after normalization, the restart probability in the success node is similar to that of the other nodes in the graph.

Definition 5 (PageRank ProPPR distribution). *A ProPPR program $\mathcal{P}$ with weights α and β defines a distribution over all answer substitutions θ for a query q as follows. Let*

$$\langle \square \circ q\theta_1 \rangle, \ldots, \langle \square \circ q\theta_k \rangle, \langle \leftarrow s_{k+1} \circ q\theta_{k+1} \rangle, \ldots, \langle \leftarrow s_m \circ q\theta_m \rangle$$




```
0.3: q(X) ← r(X), s(X).
0.7: q(X) ← u(X).

0.6: r(a).
0.4: r(b).

0.3: s(a).
0.7: s(b).

0.1: u(a).
0.9: u(b).
```

Fig. 3. PageRank ProPPR graph with restart weight $\alpha = 0.25$ and loop weight $\beta = 1$ derived from the SSLD tree in Fig. 1, with its corresponding program.

be an enumeration of all nodes in the PageRank ProPPR graph $\mathcal{G}_{\mathcal{P},\alpha,\beta,q}$, and $\pi = (\pi_1, \ldots, \pi_m)$ the PageRank vector defined by the graph. The PageRank ProPPR distribution is given by the vector $P_{\mathcal{P},\alpha,\beta}(q) = (P_1, \ldots, P_k)$ whose entries correspond to the normalized weight of the success nodes, i.e.,

$$P_i = \frac{\pi_i}{\sum_{j=1}^{k} \pi_j}$$

The PageRank ProPPR distribution for the example in Figure 4 is (P_a, P_b) with

$$P_a = \pi_{\langle \square \circ q(a) \rangle} / (\pi_{\langle \square \circ q(a) \rangle} + \pi_{\langle \square \circ q(b) \rangle}) = 0.19$$
$$P_b = \pi_{\langle \square \circ q(b) \rangle} / (\pi_{\langle \square \circ q(a) \rangle} + \pi_{\langle \square \circ q(b) \rangle}) = 0.81.$$

3 Personalized PageRank over the SSLD Tree

Compared to the SSLD tree, the PageRank ProPPR graph adds restart edges on all nodes, adds self-loops on leaves, and omits failure nodes. We now discuss the effect of performing personalized PageRank over the SSLD tree with (a) both restart edges and self-loops added, and (b) only restart edges added. The discussion of failure nodes is deferred to Section 4.

As in the case of ProPPR, the additional edges specify the possible jumps, which allows us to incorporate the jump directly into the transition probabilities instead of using a personalization vector.

Definition 6 (PageRank SSLD graphs). *For SLP $\mathcal{S}$, query q and restart probability $\gamma \in [0, 1]$, the non-loopy PageRank SSLD graph $\mathcal{G}_{\mathcal{S},\gamma,q}$ consists of the nodes and edges of the SSLD tree $\mathcal{T}_{\mathcal{S}}$ and an additional edge from every node to the root. An edge labeled p_i in $\mathcal{T}_{\mathcal{S}}$ is labeled $p_i \cdot (1 - \gamma)$ in $\mathcal{G}_{\mathcal{S},\gamma,q}$, edges from*

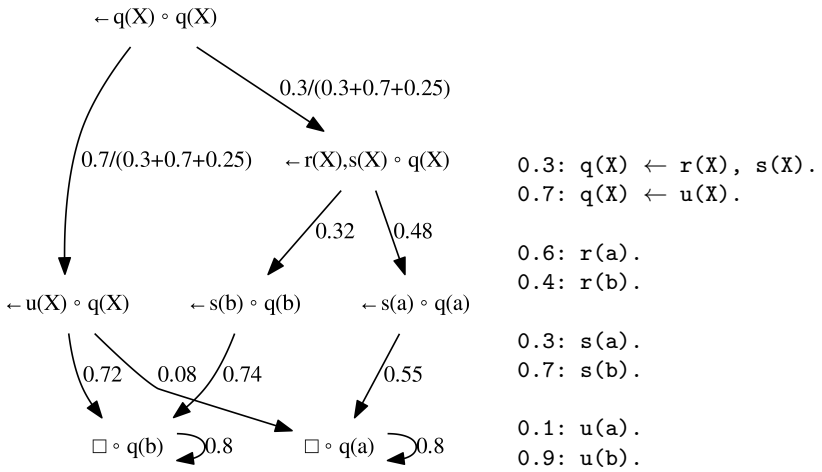


Fig. 4. The ProPPR example of Fig. 3 with transition probabilities as edge labels, omitting restart edges (labeled with the remaining probability mass at each node).

leaves to the root are labeled 1*, and all other restart edges are labeled* γ*. The loopy PageRank SSLD graph* $\mathcal{G}^{\text{loop}}_{\mathcal{S},\gamma,q}$ *is* $\mathcal{G}_{\mathcal{S},\gamma,q}$ *extended with a direct self-loop for every leaf (*$\langle\Box \circ q\theta\rangle$ *or* failure*) of* $\mathcal{T}_{\mathcal{S}}$*. An edge labeled* p_i *in* $\mathcal{T}_{\mathcal{S}}$ *is labeled* $p_i \cdot (1-\gamma)$ *in* $\mathcal{G}^{\text{loop}}_{\mathcal{S},\gamma,q}$*, self-loops are labeled* $1-\gamma$*, and all restart edges are labeled* γ*.*

Note that all edge labels take values in $[0, 1]$ here, and for every node, labels of outgoing edges sum to 1, i.e., edge labels directly correspond to transition probabilities. Figure 5 shows the loopy PageRank SSLD graph for the SLP and query in Figure 1 with $\gamma = 0.2$, where we omit restart edges to avoid clutter.

Definition 7 (PageRank SSLD distribution). *A (loopy or non-loopy) PageRank SSLD graph* $\mathcal{G}$ *defines a PageRank vector* π *with an entry* π_N *for every node* N *in the graph. For a query* q *with answer substitution* θ*, we obtain the PageRank SSLD probability*

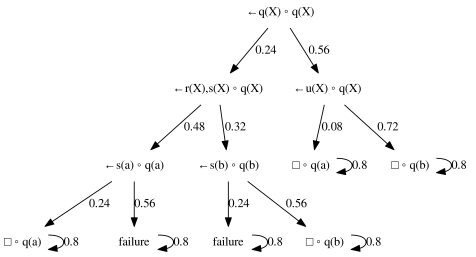
$$P_{\mathcal{G}}(q\theta) = \frac{\sum_{N \in \mathcal{N}_\theta} \pi_N}{\sum_{M \in \mathcal{N}} \pi_M}$$

where $\mathcal{N}_\theta$ *is the set of success nodes with answer substitution* θ*, and* $\mathcal{N}$ *the set of all success nodes.*

For our example SLP $\mathcal{S}$ in Figure 1, we have

$$P_{\mathcal{G}}(\mathtt{q(a)}) = \frac{0.027648 + 0.0448}{0.027648 + 0.0448 + 0.4032 + 0.043008} = 0.1397$$
$$P_{\mathcal{G}}(\mathtt{q(b)}) = \frac{0.4032 + 0.043008}{0.027648 + 0.0448 + 0.4032 + 0.043008} = 0.8603$$

These differ from the SLP probabilities $P_{\mathcal{S}}(\mathtt{q(a)}) = 0.148$ and $P_{\mathcal{S}}(\mathtt{q(b)}) = 0.852$, but can be obtained from a modified, incomplete SLP that scales the probabilities in $\mathcal{S}$ to account for the restart (shown on the right of Figure 5 for our example).



0.24: q(X) ← r(X), s(X).
0.56: q(X) ← u(X).
0.20: q(X) ←$\mathcal{R}$.

0.48: r(a).
0.32: r(b).
0.20: r(X) ←$\mathcal{R}$.

0.24: s(a).
0.56: s(b).
0.20: s(X) ←$\mathcal{R}$.

0.08: u(a).
0.72: u(b).
0.20: u(X) ←$\mathcal{R}$.

0.24: q(X) ← r(X), s(X).
0.56: q(X) ← u(X).

0.48: r(a).
0.32: r(b).

0.24: s(a).
0.56: s(b).

0.08: u(a).
0.72: u(b).

Fig. 5. The loopy PageRank SSLD graph (restart edges omitted) with restart probability 0.2 for the SLP in Fig. 1, an explicit notation for that SLP under the PageRank interpretation (middle) and the corresponding incomplete SLP (right).

Theorem 1. *For a given SLP $\mathcal{S}$, query q and restart probability γ, let $\mathcal{S}_\gamma$ be the incomplete SLP obtained by multiplying all labels in $\mathcal{S}$ with $1-\gamma$. For every answer substitution θ, we have*

$$P_{\mathcal{G}_{\mathcal{S},\gamma,q}}(q\theta) = P_{\mathcal{G}^{\text{loop}}_{\mathcal{S},\gamma,q}}(q\theta) = P_{\mathcal{S}_\gamma}(q\theta).$$

We prove the theorem by showing that PageRank converges to a vector that defines the same normalized distribution over success nodes as $\mathcal{S}_\gamma$.

Convergence It suffices to show that the PageRank SSLD Markov chain is irreducible, aperiodic, and positive-recurrent [8]. Since every state can be reached from and has a restart transition back to the query state, the chain is irreducible. Since the restart transition in the query node creates a loop, any state can be visited at an irregular time, and the chain is thus aperiodic. Given that the chain is irreducible, positive recurrency directly follows for finite state spaces (as given by finite SSLD trees), and can be shown for infinite state spaces by showing that the mean recurrence time of one state (i.e. the expected number of steps before that state is first revisited) is finite. The mean recurrence time of the query state in the loopy PageRank SSLD graph $\mathcal{G}^{\text{loop}}_{\mathcal{S},\gamma,q}$, where every state has a transition with probability γ to the query state, is given by the finite quantity

$$\text{MRT}(q) = \sum_{n=1}^{\infty} n \cdot \gamma \cdot (1-\gamma)^{n-1} = \frac{1}{\gamma}$$

For the case of $\mathcal{G}_{\mathcal{S},\gamma,q}$, we have $\text{MRT}^{\text{loop}}(q) \leq \int_\gamma \text{MRT}(q)$. Let a be the infimum over the set of all restart transition probabilities in the Markov chain. Since $\int_{\gamma=a}^{1} \frac{1}{\gamma}$ for fixed $a > 0$ is finite, the query node is positive recurrent.

Equivalence of distributions over success nodes We first consider the non-loopy graph $\mathcal{G}_{\mathcal{S},\gamma,q}$. Let π_q be the value associated with the query node in the corresponding PageRank vector π, and π_N the value associated with the success node reached by the SSLD refutation $p_1 : C_1, \ldots, p_n : C_n$, where the p_i are the labels in $\mathcal{S}$. The corresponding labels in $\mathcal{S}_\gamma$ are thus $p'_i = p_i \cdot (1 - \gamma)$. We have

$$\pi_N = \pi_q \cdot \prod_{i=1}^{n} p'_i$$

which is the probability of the corresponding refutation in $\mathcal{S}_\gamma$ multiplied by the constant π_q. When normalizing over all successful derivations, this constant cancels out, and we thus obtain the same distribution as for $\mathcal{S}_\gamma$. Similarly, for the loopy graph $\mathcal{G}^{\text{loop}}_{\mathcal{S},\gamma,q}$, we have

$$\pi_N = \pi_q \cdot \prod_{i=1}^{n} p'_i + (1 - \gamma) \cdot \pi_N = \frac{\pi_q}{\gamma} \cdot \prod_{i=1}^{n} p'_i$$

which again only differs by a multiplicative constant from the probability of the corresponding refutation in $\mathcal{S}_\gamma$. □

The effect of the restart can also be described by the logic program labeled with PageRank SSLD probabilities in the middle of Figure 5. The special symbol $\mathcal{R}$ for the jump to the query node highlights the extra-logical nature of the restart step, which is similar to Prolog's cut operator `!/0`, but has a far greater effect.

Prolog implicitly builds an SLD tree by applying SLD resolution in a depth-first manner. Given a clause $b \leftarrow a_1, \ldots, a_m, !, \ldots, a_n$ Prolog commits to the clauses selected during the application of SLD resolution on the literals $b, a_1, \ldots, a_m$ when reaching the cut, i.e., Prolog does not consider any alternative clauses. This is equivalent to building the SLD tree for a program where cuts have been omitted, and then pruning away all alternative branches starting at the nodes associated with these literals. This pruning causes the search space to be reduced and results in the potential loss of solutions. Instead of directly pruning away alternative clauses, the restart clause causes the query to be stochastically restarted. Resolution on a goal such as ($\leftarrow$ `r(X)`, `s(X)`) using the restart clause for the first subgoal `r(X)` results in the initial query with fresh variables, i.e., all subsequent subgoals (here just `s(X)`) are dropped, and all substitutions obtained so far forgotten. The restart affects the semantics by reducing the probability associated with all clauses for its predicate. When considering approximate inference (see Section 4) where branches are pruned when their associated probability drops below a particular threshold, we observe that this also results in branches being pruned earlier.

4 Discussion

We now discuss the differences between the transition probabilities used by PageRank SSLD and PageRank ProPPR, respectively. The types of nodes we

need to consider are success nodes labeled $\langle \square \circ q\theta \rangle$, failure nodes, and inner nodes labeled $N = \langle \leftarrow g_1, \ldots, g_m \circ q\theta \rangle$ with the predicate of g_1 defined in either $\mathcal{C}_p$ or $\mathcal{C}_{db}$. For such nodes N, let $\{w_1 : C_1, \ldots, w_k : C_k\}$ be all clauses in the program for which the head unifies with g_1, and $\{w_{k+1} : C_{k+1}, \ldots, w_n : C_n\}$ all those whose head has the same predicate as g_1, but does not unify with g_1.

For PageRank SSLD, $\sum_{i=1}^{n} w_i = 1.0$, and there is no distinction between $\mathcal{C}_p$ and $\mathcal{C}_{db}$. Based on Theorem 1, we only consider the loopy case here. Both success and failure nodes then have a restart probability of γ, and a self-loop probability of $1 - \gamma$. For an inner node N, the transition probabilities are γ for the restart edge, and $(1 - \gamma) \cdot w_i$ for the edge using clause C_i.

It is easy to verify that the same transition probabilities would be achieved by normalization over all outgoing edges with initial weight $\alpha = \frac{\gamma}{1-\gamma}$ on the restart edge, and $\beta = \sum_{i=1}^{n} w_i = 1$ on the self-loops. As our PageRank SSLD example uses $\gamma = 0.2$, we set $\alpha = 0.2/0.8 = 0.25$ and $\beta = 1$ in the corresponding PageRank ProPPR example. In the absence of failure edges ($n = k$ for all inner nodes), and with $\mathcal{C}_{db} = \{\}$ (as in SLPs), this choice of restart and self-loop weights makes the two semantics coincide.

In general, however, the transition probabilities of PageRank ProPPR depend on the failure nodes in the SSLD tree, as in inner nodes, the normalization does not include the weights of clauses for which unification fails.

If the first subgoal g_1 of inner node N is defined in $\mathcal{C}_{db}$, the restart probability in N is α, and the transition probability for each clause C_i with $1 \leq i \leq k$ is $p_i^p = (1 - \alpha)/k$, whereas the latter would be $p_i^s = (1 - \alpha)/n$ if normalized over all clauses as in PageRank SSLD. Thus, we have $p_i^p = p_i^s \cdot (n/k)$.

In the case of $\mathcal{C}_p$, the restart probability is $\alpha/(\alpha + \sum_{j=1}^{k} w_j)$, and the transition probability of such a C_i's edge is $p_i^p = w_i/(\alpha + \sum_{j=1}^{k} w_j)$. That is, the transition probability for a clause's edge increases as competing clauses fail. This effect can be quantified as follows:

$$p_i^p = \frac{\alpha + \sum_{j=1}^{k} w_j + \sum_{j=k+1}^{n} w_j}{\alpha + \sum_{j=1}^{k} w_j} \cdot p_i^s = (1 + \frac{\sum_{C_j \text{fails}} w_j}{\alpha + \sum_{C_j \text{succeeds}} w_j}) \cdot p_i^s$$

where $p_i^s = w_i/(\alpha + \sum_{j=1}^{k} w_j + \sum_{j=k+1}^{n} w_j)$ is the transition probability with normalization over all clauses as in PageRank SSLD.

As an illustration, consider the node labeled $\langle \leftarrow s(a) \circ q(a) \rangle$ in our running example. In the PageRank SSLD distribution defined by the program in Figure 5, the probability of the edge associated with clause $s(a)$ is 0.24. In the PageRank ProPPR distribution in Figure 4, the probability of this edge is $0.55 = (1 + \frac{0.7}{0.3+0.25}) \cdot 0.24$. Similarly, the probability of the edge below node $\langle \leftarrow s(b) \circ q(b) \rangle$ increases from 0.56 to 0.74. This also illustrates that the weights in the ProPPR program cannot directly be interpreted as the relative importance of different answers to a predicate, as answers with lower weight in the program may get a higher increase in transition probability if other clauses fail.

To summarize, each transition probability in the Markov chain given by a ProPPR program depends not only on the weight of the associated clause and

the restart weight α, but also on the weight of the failing competing clauses. Because of this additional dependency, it is not possible to concisely represent a ProPPR program as an SLP. On the other hand, for a complete SLP whose SSLD tree contains no failure edges, the PageRank SSLD distribution with restart probability γ coincides with the PageRank ProPPR distribution with restart weight $\alpha = \frac{\gamma}{1-\gamma}$ and self-loop weight $\beta = 1$. In general, however, the different approaches to derive transition probabilities lead to different distributions.

The different restart probabilities of the nodes in the PageRank ProPPR graph also play a role in ProPPR's approximate inference algorithm [9], which is derived from the method for approximating personalized PageRank vectors used in the PageRank-Nibble algorithm [1]. This method assumes the personalized PageRank setup with transition matrix S, fixed jump probability γ and personalization vector s as discussed in Section 2.1. We discuss the implications in the context of ProPPR below, but first summarize the approach itself.

The algorithm has a stopping parameter $\epsilon \in (0, 1]$ and maintains two vectors p (the approximation of the PageRank vector) and r (the residual vector), each with an entry for every node in the graph. Initially, all entries in p are zero, and r is a distribution over nodes, which in the cases of interest to our discussion puts the full mass on the query node. Let $n(N)$ be the number of outgoing edges of node N, and p_i the transition probability from N to N_i. While there is a node N with $r(N) \geq \epsilon \cdot n(N)$, for one such N, the following updates are performed:

$$\begin{aligned} p(N) &= p(N) + \gamma \cdot r(N) \\ r(N_i) &= r(N_i) + (1-\gamma) \cdot p_i \cdot r(N) \\ r(N) &= 0 \end{aligned}$$

In time $O(\frac{1}{\gamma \cdot \epsilon})$, the algorithm converges to an approximation p of the PageRank vector where the error for every node N is bounded by $\epsilon \cdot n(N)$, considering no more than $\frac{1}{\gamma \cdot \epsilon}$ transition edges, and thus has a complexity independent of the size of the full graph (and its underlying program). Furthermore, this approach allows to (implicitly) construct the graph on the fly, expanding the vectors to new nodes as they first appear in the updates.

For PageRank SSLD, this algorithm can directly be applied with the loopy graph, by setting s to the vector with all mass on the query node, and S to the transition matrix with self-loop transition probabilities of 1 and the transition probabilities for all edges as given in the SSLD tree. Because of the tree structure (with loops on the leaves only) underlying the transition matrix, the stopping criterion corresponds to pruning the SSLD tree below nodes N with $P_{\mathcal{S}_\gamma}(N) < \epsilon \cdot n(N)$ with $n(N)$ the total number of outgoing edges (including the restart).

For PageRank ProPPR, constructing the transition matrix and restart vector is more involved, as there is no constant restart probability γ, and furthermore, the values of restart probabilities are not known a priori. ProPPR therefore uses a constant γ that is expected to provide a lower bound on these values as the jump probability, sets s to the vector with all mass on the query node as in the SSLD case, and modifies the transition matrix S discussed for the SSLD case to also take into account the difference between the actual restart probabilities

and the lower bound γ. In contrast to the SSLD case, this transition matrix corresponds to a graph with longer loops, where the stopping criterion is not directly expressible in terms of the derivation probability. It does however allow ProPPR to benefit from the advantages of the approximation method.

5 Conclusions

We have adopted ProPPR's view of probabilistic inference as a random walk over a graph constructed from a labeled logic program to investigate the relationship between ProPPR and SLPs. We have shown that the distribution obtained from a random walk with restart over the SSLD tree of a pure and complete SLP coincides with the distribution defined by an incomplete SLP with the same clauses, but with labels rescaled to take into account the restart probability. We have further shown that the differences in semantics rule out direct, generally applicable translations between ProPPR programs and SLPs.

Acknowledgements Dries Van Daele is supported by PF/10/010 (NATAR). A. Kimmig is supported by the Research Foundation Flanders (FWO).

References

1. Reid Andersen, Fan Chung, and Kevin Lang. Using PageRank to locally partition a graph. *Internet Mathematics*, 4(1):35–64, 2007.
2. Prasad Chebolu and Páll Melsted. PageRank and the random surfer model. In *Proceedings of the 19th annual ACM-SIAM symposium on Discrete algorithms*, 2008.
3. James Cussens. Stochastic logic programs: Sampling, inference and applications. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, 2000.
4. Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors. *Probabilistic Inductive Logic Programming — Theory and Applications*, volume 4911 of *Lecture Notes in Artificial Intelligence*. Springer, 2008.
5. Lise Getoor and Ben Taskar, editors. *An Introduction to Statistical Relational Learning*. MIT Press, 2007.
6. Stephen Muggleton. Stochastic logic programs. *Advances in inductive logic programming*, 32:254–264, 1996.
7. Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.
8. Sidney I. Resnick. *Adventures in Stochastic Processes*. Birkhauser Verlag, 1992.
9. William Yang Wang, Kathryn Mazaitis, and William W Cohen. Programming with personalized PageRank: a locally groundable first-order probabilistic logic. In *Proceedings of the 22nd ACM International Conference on information & knowledge management*, 2013.
10. William Yang Wang, Kathryn Mazaitis, and William W. Cohen. ProPPR: Efficient first-order probabilistic logic programming for structure discovery, parameter learning, and scalable inference. In *AAAI Workshop on Statistical Relational Artificial Intelligence (StaRAI 14)*, 2014.